

PRESENTATION SERVER

Inventors

Eric D. Bloch

Max D. Carlson

Christopher Kimm

James B. Simister

Oliver W. Steele

David T. Temkin

Adam G. Wolff

"Express Mail" mailing label number: EL 795 247 947 US

PREPARED BY
VIERRA MAGEN MARCUS HARMON & DENIRO LLP
CUSTOMER ID: 000028554

PRESENTATION SERVER

5 This application claims the benefit of U.S. Provisional Application No. 60/349,671, entitled, "Interactive System," filed on January 16, 2002, incorporated herein by reference.

CROSS-REFERENCE TO RELATED APPLICATIONS

10 This Application is related to the following Application: "Interface Engine Providing A Continuous User Interface," by J. Bret Simister, Adam G. Wolff, Max D. Carlson, Christopher Kimm, and David T. Temkin, Attorney Docket No. LZLO-01002US0, filed the same day as the present application and incorporated herein by reference.

BACKGROUND OF THE INVENTION

Field of the Invention

15

The present invention is directed to technology for providing content.

Description of the Related Art

20 The Internet has become very popular. Many people and businesses use the Internet to access data, applications, entertainment, productivity tools, and so on. However, web pages and web applications are built around the page metaphor, which constrains application interaction. That is, typical web pages are displayed a page at a time and lack the fluid feel and features of other user interfaces. Many feel that this limitation is due to the shortcomings of HTML, while others lay blame on the
25 limitations of browsers.

Some previous attempts to provide dynamic user interactions include downloading source code directly to the client. For example, Javascript code is often downloaded directly to the browser. The browser then interprets the code directly. This approach is limited because Javascript is typically not portable (different

browsers offer differing support for the language) and Javascript severely limits the interactions that can be specified by the programmer.

Another attempted solution involves the downloading of statically pre-compiled bytecode. For example, Java applets can be downloaded to a browser when a user accesses a web page. The Java source code for the applet is fixed, written in advance, and pre-compiled into an applet. When the user accesses the page, the applet is downloaded into the browser and the bytecode is interpreted (or compiled into machine code by a JIT compiler). One of several shortcomings of this approach is that Java is not sufficiently prevalent on all types of clients. That is, not all clients have the ability to interpret (or JIT compile) Java bytecode. Another shortcoming is that applets can be quite large to download and slow to run. Furthermore, creating Java applets is a complicated processes. Many web designers are not able or do not have the resources to take advantage of Java applets. Finally, because the code is written and compiled in advance, it is not dynamic.

Another attempt to provide more complex user interactions includes developing a complete standalone (non-browser based) application. The source code for this application is compiled statically. The end-user must specifically request that the binary object code for the program be downloaded over the network or he/she must install it from some other external source. This approach is typically even more expensive than Java applets and suffers even worse deployment issues.

Another solution includes dynamically generated HTML, which involves dynamically creating HTML source code in response to a request for content. The dynamically created source code is then transmitted from the server to the client browser, where it is used to provide content to the user. Although the code is generated dynamically, the code is still HTML which limits, as described above, the features of the user interface.

Thus, there are no complete solutions today that provide complex, feature rich dynamic interactions without the shortcomings of the prior art.

SUMMARY OF THE INVENTION

5 The present invention, roughly described, includes a system that presents content (e.g. including, but not limited to, interactive content, applications, video, audio, etc.) to a client without the shortcomings of the prior art. One embodiment of the present invention includes a server that receives a request for particular content. The server accesses a mark-up language description of the particular content and compiles that mark-up language description to create executable code (e.g. object code, byte code, etc.). The compilation of the mark-up description by the server is performed in response to the received request. The executable code is then transmitted from the server to the client. In one embodiment, the server can be contained within an application server or web server, and the client includes a browser that communicates with the server via the Internet, or other network.

10 Another embodiment of the present invention also includes a server receiving a request for particular content. In response to the request, the server accesses first code associated with the particular content and compiles the first code to create executable code which implements a user interface that provides access to the particular content. The server then transmits the executable code to the client. The user interface provides windows, dialogs, buttons, images, sounds, text fields, banners, animation, etc.

15 In some embodiments, the requested content includes data. The requested data is accessed at the server and compiled to create executable code. The executable code includes a representation of the data. The server transmits the executable code to the client.

20 By compiling the code at the server, the downloads to the client are smaller and the client does not need to take the time to compile the downloaded code. Therefore, the user experiences less time waiting for the interaction with the desired content.

25 The present invention can be accomplished using hardware, software, or a combination of both hardware and software. The software used for the present invention is stored on one or more processor readable storage devices including hard

disk drives, CD-ROMs, DVDs, optical disks, floppy disks, tape drives, RAM, ROM or other suitable storage devices. In one embodiment, the software can be performed by one or more processors. In alternative embodiments, some or all of the software can be replaced by dedicated hardware including custom integrated circuits, gate arrays, FPGAs, PLDs, and special purpose processors. One example of hardware that can implement all or portions of the present invention includes a processor, storage elements, peripheral devices, input/output devices, displays, and communication interfaces, in communication with each other as appropriate for the particular implementation.

The advantages of the present invention will appear more clearly from the following description in which the preferred embodiment of the invention has been set forth in conjunction with the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of one embodiment of the present invention.

Figure 2 is a block diagram of a second embodiment of the present invention.

Figure 3 is a flow chart describing one embodiment of a process for implementing the present invention.

Figure 4 is a block diagram of a presentation server according to one embodiment of the present invention.

Figure 5 is a flow chart describing one embodiment of the operation of the presentation server.

Figure 6 is a block diagram of one embodiment of a compiler.

Figure 7 is a flow chart describing one embodiment of the operation of the compiler.

Figure 8 is a flow chart describing one embodiment of a process for compiling media.

Figure 9 is a flow chart describing one embodiment of a process for compiling view instances.

Figure 10 is a flow chart describing one embodiment of a process for performing at least a portion of the executable code.

Figure 11 is a flow chart describing one embodiment of a process for responding to a data connection request.

5 Figure 12 is a flow chart describing one embodiment of a process for compiling data.

DETAILED DESCRIPTION

10 The presentation server includes server-deployed software that enables developers to easily and economically create network-aware applications with rich, interactive user-interfaces. The presentation server is designed to receive a mark-up language description of a user-interface and dynamically compile that mark-up language description to executable code. In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services. It contains standard user interface primitives like "window," "button," "text," "scroll bar," and so on, as well as syntax for automatically connecting user-interface items with back-end data sources and services. The mark-up language can also include a scripting language for procedural specification of application behavior that is similar to Javascript.

15 In one embodiment, the presentation server generates highly optimized/compressed object code for a given Presentation Renderer. A Presentation Renderer is a software environment, hardware, set of one or more software programs, etc. that can display graphics and play sound. Figure 1 is a block diagram providing a high-level description of one exemplar implementation of the present invention. Figure 1 shows Presentation Server 10 in communication with external data source(s) 12 and client Presentation Renderer 14. Communication between the various elements can be via network, dedicated connection, wireless connection, or any other connection that is suitable for appropriate communication. The external data sources 20 can include databases, directories, web services, or any other storage device or

structure that can hold data. Client Presentation Renderer 14 can be generic software for providing a user-interface or can be specific software for the purpose of communicating with Presentation Server 10.

In one embodiment, client Presentation Renderer 14 is a Macromedia Flash Player embedded in a web client as a plug-in. While the Flash Player is an appropriate vehicle for Presentation Server 10, there is nothing inherent in the design of Presentation Server 10 that requires the Flash Player. Many other presentation renderers can also be utilized.

Figure 2 is a block diagram of another embodiment of the present invention. Figure 2 shows Presentation Server 50 implemented as a Java Servlet that compiles server located mark-up language description files and data into object code. In one embodiment, Presentation Server 50 generates object code for the Macromedia Flash Player. Presentation Server 50 can be hosted by any standard Java Servlet implementation, including Jakarta Tomcat and J2EE servers like BEA Weblogic and IBM Websphere. When hosted in a J2EE server, the presentation server takes advantage of services that are available including JDBC and JCA. Figure 2 shows Presentation Server 50 hosted in Application Server 52. Application server 52 also includes JDBC to RDBMS services 54, which is in communication with relational database 56. As stated above, other types of data sources, other than a relational database can also be used. Presentation Server 50 receives requests and sends responses via Web Server 58, which is in communication with clients via a network. That network can be any standard network known in the art, including the Internet, a LAN, a WAN, etc. For example, Figure 2 shows an HTTP client 60 (e.g. browser) with plug-in 62 (e.g. Flash Player) in communication with Presentation Server 50 via Web Server 58.

Figure 3 is a flow chart describing one exemplar embodiment of the operation of the present invention. In step 102, the Presentation Server receives a request for content. The content can be a Web page, media, data, an application, or anything else accessible via a network. In step 104, the Presentation Server accesses a mark-up language description of the content in response to the received request. In some

embodiments, a mark-up language description is not used and, thus, step 104 is not performed. In step 106, Presentation Server accesses any data and/or media in response to the request or in response to the mark-up language description. In some instances, the request will not require any data or media and, therefore, step 106 can be skipped. In step 108, the mark-up language description, data and/or media are compiled to create executable code for a user interface. In step 110, the executable code is transmitted to the client Presentation Renderer. In step 112, the executable code is executed at the client Presentation Renderer.

As stated above, some embodiments include the Presentation Server accessing a mark-up language description of the content requested. There are many different variations of mark-up languages that are suitable for the present invention. XML provides one example of a suitable framework for a mark-up language. One embodiment of the present invention uses an application of XML, referred to as the Laszlo application language. The Laszlo application language is described below.

Content available to the Presentation Server is described by one or more Laszlo application language source files. This can be a single file which defines all aspects of an application, or a source file can reference other files through the use of include statements. The Laszlo application language is an application of XML, with ECMAScript used to specify user interface behaviors and application logic. ECMAScript code is contained within script tags, or as the value of certain attributes. This is similar to the use of JavaScript to enhance HTML pages, except that the document specifies the view structure of an application, rather than the text structure of a page. Features of the Laszlo application language include: the ability to define user-interface elements via XML, or ECMAScript; the ability to define custom user-interface classes, incorporating custom art assets and behaviors, via XML, or ECMAScript; the ability to attach custom properties and behaviors to user-interface elements and classes; XML access to all classes in the runtime support library; the ability to include external media assets such as JPEG and PNG images, SWF animations, and MP3 audio; inline definition of formatted text (via HTML), and

vector graphics (via SVG); data source connections to define a connection to an external data source; and support for multiple skins.

Below is an example of Laszlo application language:

```
5 <lz>
  <window title="Hello World">
    <text><i>Hello</i>, World!</text>
    <button onclick="beep()" ">Press Me!</button>
  </window>
10 <script>
    function beep() {sys.beep()}
  </script>
</lz>
```

15 When the above code is executed, a single window titled "Hello World" is displayed. The window includes the text "Hello, World!" (with "Hello" in italics) and a button which plays a beep sound when pressed.

A Laszlo application language source file is an XML file, with a root element of lz. The lz element contains elements that define various aspects of the application: its visual appearance, its behavior, and its connection to an external data source. Examples of the different elements include:

- View templates;
- View class definitions;
- 25 • Scripts, for program logic; and
- Database connection definitions.

Below is an example of code that uses these elements. The code will become clearer to the reader as the discussion continues.

```
<lz>
  <window title="Simple Window" x="5" y="10">
    <image src="logo.jpeg"/>
    <text>Hello, World!</text>
5    <button onclick="otherWindow.show();">
      Click Me!</button>
  </window>
  <class name="MyWindow" extends="LzWindow">
    ...
10  </class>
  <window id="otherWindow" class="MyWindow"
visible="false">
    <svg>
      <rect x="100" y="50" fill-color="red"/>
15      <circle x="100" y="50" fill-color="green"/>
    </svg>
    <button onclick="userfn()" />
  </window>
  <script>
20    def userfn() {otherWindow.hide()}
  </script>
</lz>
```

25 A view template is an XML description of a view (a window, a button, a dialog, or other user interface element), which is instantiated when the application is run. A view template may occur immediately within the document root element, or nested within another view template (or a class definition). A view template that occurs directly within the document root element is a root view template. An application source file may contain any number of root view templates, and they may

30 be interleaved with other elements, such as script, data source elements and class definitions. If an application doesn't contain any view templates, it will have no

visible behavior, unless its script code uses the view system API to create a view programmatically.

5 An application source file may contain at most one canvas element. If a canvas element is present, it must precede (or contain) any other view elements. The canvas element defines properties of the global canvas, which is the root of the runtime view hierarchy; all other views are directly or indirectly contained within the canvas element. If the canvas element is not present, an implicit canvas is created. All root view templates create views that are children of the canvas, whether it is explicit or not.

10 A view template may contain other view templates. The XML element hierarchy is mapped to a runtime view hierarchy: view templates that are children of a parent view template instantiate views that are children of the parent view. Below is an example of application source file that defines a single view:

15 <lz>
 <window title="Title" x="5" y="10"/>
</lz>

Below is an example of pseudocode that defines nested views

20 <lz>
 <window title="Title" x="5" y="10">
 <view .../>
 <view .../>
 </window>
25 </lz>

30 Each view is an instantiation of a view class. A view class supplies default properties, behavior, and child views, which specific view templates instantiate. The system includes pre-defined view classes and user defined view classes (view class definitions).

The following is an exemplar list of pre-defined view classes: canvas, dialog, window, view, listView (a vertical list of items), tableView (a horizontal list of listViews), gridView, menubar, menu, menu item, button, and text.

The table below lists XML elements and their ECMAScript equivalents:

XML element name	ECMAScript class name
canvas	LzCanvas
dialog	LzDialog
window	LzWindow
text	LzText
button	LzButton
menubar	LzMenubar
menu	LzMenu
menuitem	LzMenuItem
instance	user-defined view class

5

Each view template includes a set of attributes. The table below lists exemplar attributes.

Attribute name	Type	Default value	Description
title	string	""	Title (for windows and dialogs)
visible	boolean	true	If true, the view is visible when initially instantiated.
x	float	0.0	Horizontal displacement from the containing view, or from the canvas.
y	float	0.0	Vertical displacement from the containing view, or from the canvas.

A view instance may also contain one or more references to media. One example of a reference to media is an image reference. An image reference is a reference to an external image, or an embedded svg element. SVG is an XML application for describing vector graphics. SVG is known in the art and more information about SVG can be found at www.w3.org. The image is usually used as a background image, but the interpretation of an image reference contained within a view depends on the view that contains it. In one embodiment, an image reference is created in one of at least three ways:

- (1) The svg element contains inline svg.

```
<window>
  <svg>
    <rect width="100" height="50" fill-color="red" frame-
color="blue"/>
    <circle width="100" height="50" fill-color="green"/>
  </svg>
</window>
```

- (2) The image element references an external media file that contains a static image.

```
<window>
  <image src="image.jpeg"/>
  <button>Okay</button>
</window>
```

- (3) The element references a swf file, that contains a Flash movie.

```
<window>
  <swf src="image.swf"/>
```

```
<button>Okay</button>
</window>
```

5 Within the Laszlo application language, it is possible to associate an action with events that occur when a user interacts with an application. The attributes in the list below are associated with scripts that are executed when the corresponding event occurs. The scripts are created by the developer who creates the mark-up language description. The names of the attributes, and the associated events, are defined in HTML 4.01 Specification.

name	description
onclick	The onclick script is executed when the pointing device button is clicked over an element.
ondblclick	The ondblclick script is executed when the pointing device button is double clicked over an element.
onmousedown	The onmousedown script is executed when the pointing device button is pressed over an element.
onmouseup	The onmouseup script is executed when the pointing device button is released over an element.
onmouseover	The onmouseover script is executed when the pointing device button is moved onto an element.
onmousemove	The onmousemove script is executed when the pointing device is moved while it is over an element.
onfocus	The onfocus script is executed when an element receives focus either by the pointing device or by tabbing navigation.
onblur	The onblur script is executed when an element loses focus either by the pointing device or by tabbing navigation.
onkeypress	The onkeypress script is executed when a key is pressed and released over an element.
onkeydown	The onkeydown script is executed when a key is pressed down over an element.
onkeyup	The onkeyup script is executed when a key is released over an element. This attribute may be used with most elements.
onselect	The onselect script is executed when a portion of text is selected.
onchange	The onchange script is executed when text is changed.

The application file may contain view class definitions, which use XML to define view classes. A view class definition is created using the following exemplar syntax:

```
5  <class name="MyWindow" extends="LzWindow">
    <property name="p1" value="1"/>
    <property name="p2" value="'2'"/>
    <view ...>
    <view ...>
10 </class>
```

This code declares a class named MyWindow that subclasses LzWindow, has two additional properties p1 and p2 with default values 1 and 2, and has two default child views, in addition to whatever default child views LzWindow has. A view class definition can also override methods of a super class. This example could also have been written as:

```
15 <script>
    class MyWindow extends LzWindow {
20     val p1=1;
        val p2='2';
        constructor MyWindow() {
            this.attach(...);
            this.attach(...);
25     }
    }
</script>
```

Script code can instantiate the class as follows:

30

```
w = new MyWindow(...).
```

A view template can define an instance of the new class thus:

```
5  <window class="MyWindow" x=...>
    <property name= . . . value= . . ./>
    <property name= . . . value= . . ./>
    <view ...>
    </window>
```

10 The above described mark-up language can be used to describe many different user interfaces. The exact type of user interface is not important to the present invention. One example of a user interface system can be found in the the Application: "Interface Engine Providing A Continuous User Interface," by J. Bret Simister, Adam G. Wolff, Max D. Carlson, Christopher Kimm, and David T. Temkin, Attorney Docket No. LZLO-01002US0, filed the same day as the present application and incorporated herein by reference.

15 The above discussion describes an exemplar mark-up language. This mark-up language can make use of a scripting language. For example, the Laszlo application language uses a scripting language called LaszloScript, which is based on ECMA Script-262 Edition 3, with two extensions from ECMA Script-262 Edition 4: classes definitions and the "super" expressions. Classes are defined using the ECMA Script Edition 4 class keyword. The syntax is:

```
20
25 ClassDefinition: "class" Identifier [Inheritance] Block
    | "class" Identifier
    Inheritance: empty
    | "extends" ClassName
```


In one embodiment, a LaszloScript class definition is equivalent to an XML class definition of the same symbol. In particular, the "inherits" attribute of an XML class element may refer to a class defined using the LaszloScript class definition; an XML instance element may instantiate a class defined in LaszloScript; and a LaszloScript class definition may extend a class defined in XML. A class must be defined before it is referenced. Super expressions may only be used inside a class method. Super expressions change the behavior of the operator in which it is embedded by limiting its property search to definitions inherited from the superclass.

Below is a table, which describes differences between Laszlo Script and ECMA Script3.0:

String-to-Boolean conversion	In ECMA-262, any nonempty string converts to true. LaszloScript converts strings to true only if they evaluate to a nonzero number.
Case sensitivity	In LaszloScript, keywords are case-sensitive but identifiers are not. This does not comply with ECMA-262, which demands complete case sensitivity.
Undefined datatype conversion	LaszloScript converts undefined to the empty string ("") in a string context or to 0 (zero) in a numeric context. ECMA-262 converts undefined to the string "undefined" in a string context or to NaN in a numeric context.
Unicode support	LaszloScript supports the multi-byte character sets, but does not offer full Unicode support.
eval support	The argument to eval must be a string that names a single identifier.

There are at least four mechanisms for including scripts in a Laszlo application language description of content:

- As the content of a method element. A script in this position must have the syntactic type SourceElements (defined in ECMAScript-262 Edition 3); that is, it must be a sequence of variable definitions and statements.
- As the content of script element. A script in this position must have the syntactic type Program; that is, it must be a sequence of variable definitions, function definitions, and statements.
- As the value of an event attribute. A script in this position must have the syntactic type SourceElements.
- As the content of a file referenced by a script tag's href attribute. A script in this position must have the syntactic type Program.

The first two mechanisms put the script source within an XML element. The XML markup characters < and & must be quoted to prevent them from being parsed as XML markup. An example of a badly quoted script is "<<script>function lt(a,b) {return a < b;}</script>" This won't work because the < in a < b will be interpreted as the beginning of an XML element. Examples of acceptable scripts include "<<script>function lt(a,b) {return a < b;}</script>" and "<<script><![CDATA[function lt(a,b) {return a < b;}]]</script>"

The Laszlo application language also allows for the use of data connection descriptions, which describe a connection to a data source (e.g. relational database, directory, other type of data structure, etc.). Datasources are specified via the datasource element. A datasource can include several queries, which are referenced for elements within a view template. Below is an example of datasource connection that creates an odbc datasource named 'mydb' and queries named 'mydb.contacts' and 'mydb.contacts_desc'.

```
<datasource id="mydb" type="odbc"
DSN="DSN=oracle;UID=dbuser;PWD=dbpass" public="false"/>
```

```
<query id="contacts" query="SELECT * FROM contacts
WHERE name LIKE '%a' AND city <> '' ORDER BY name" />
<query id="contacts_desc" query="SELECT * FROM
contacts WHERE name LIKE '%a' AND city <> '' ORDER BY
name DESC" />
</datasource>
```

Below is an example of a data source connection for a second embodiment:

```
<datasource id="contacts" type="jdbc">
  <param id="username">max</param>
  <param id="password">myPass</param>
  <param id="hostname">10.10.20.4</param>
  <param id="port">3306</param>
  <param id="driver">org.gjt.mm.mysql.Driver</param>
  <dataview id="addresses">SELECT * FROM
    addresses</request>
  <dataview id="all">SELECT * FROM contacts WHERE
    ("userid" ==
  <expr>userid</expr></request>
</datasource>
```

This latter example includes dataview definitions instead of queries. A dataview retrieves all records an application will work with. Also note that the second dataview definition has <expr></expr> tags. These tags allow applications to dynamically define portions of dataview definitions – in this example the specific instance's user ID.

Below is an example of a datasource connection that creates an XML datasource named 'myxml' and a query named 'myxml.serverinf'.

```
<datasource id="myxml" type="xml" file="serverinfo.xml">
```

```
<query id="serverinfo" type="xslt">
<xsl:stylesheet version="1.0" xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text" />
    <xsl:template match="/"><xsl:apply-
        templates/></xsl:template>
    <xsl:template match="context">
        &lt;record id="<xsl:value-of select="@recordid"/>"
            username="<xsl:value-of select="@username"/>" /&gt;
        <xsl:apply-templates/>
    </xsl:template>
</xsl:stylesheet>
</query>
</datasource>
```

Below is an example of code for dynamically generating table rows:

```
<datasource id="contacts" type="jdbc">
    <param id="username">max</param>
    <param id="password">myPass</param>
    <param id="hostname">10.10.20.4</param>
    <param id="port">3306</param>
    <param id="driver">org.gjt.mm.mysql.Driver</param>
    <dataview id="addresses">SELECT * FROM
        addresses</request>
    <dataview id="all">SELECT * FROM contacts WHERE
        ("userid" ==
    <expr>userid</expr></request>

    <query id="allcontacts">
        <fields>
            <field name="displayname">
```

```

        <field name="email">
        <field name="phone">
    </fields>
</query>
5  </datasource>

<window id="w1" title="Address Book" x="10" y="50"
width="450" height="300">
    <scrolledview>
10        <view datasource="contacts" dataview="all"
            query="allcontacts">
                <layout type="simple" axis="x"
                    spacing="0"/>
                <column label="Name" copies="20"
15                    subviewrecords="next">
                        <datatext field="displayname" />
                    </column>
                <column label="Name" copies="20"
                    subviewrecords="next">
20                    <datatext field="displayname" />
                    </column>
                <column label="Email" copies="20"
                    subviewrecords="next">
                        <datatext field="email" />
25                    </column>
                <column label="Phone" copies="20"
                    subviewrecords="next">
30                    <datatext field="phone" />
                    </column>
                </view>
    </scrolledview>
```

</window>

Below is an example for creating dynamic SVG:

```
5 <view>
  <for-each name="contact" source="mydb.contacts"
    position="below" spacing="2">
    <svg>
      <text value="contact.fname"/>
10    </svg>
    </for-each>
  </view>
```

15 The above description of the Laszlo application language provides one example of a suitable mark-up language that can be used with the present invention. Other mark-up languages/formats can also be used.

20 In one embodiment, a set of one or more files using the above-described (or another) mark-up language are compiled to create executable code for a presentation renderer. One example of a presentation renderer is the Macromedia Flash Player, which uses the SWF file format for its executable code. SWF files are binary files stored as 8-bit bytes. The container format consists of a header block with the structure shown below.

25 CONTAINER FORMAT

FIELD	TYPE	COMMENT
Signature	UI8	Signature byte 1 – always 'F'
Signature	UI8	Signature byte 2 – always 'W'
Signature	UI8	Signature byte 3 – always 'S'
Version	UI8	Single byte file version
File Length	UI32	Length of entire file in bytes
Frame Size	RECT	Frame size in TWIPS
Frame Rate	UI16	Frame delay in 8.8 fixed number of frames per second
Frame Count	UI16	Total number of frames in movie

Following the header are a series of tagged data blocks. Each tag has a tag type and a length. There is both a short and long tag header format. Short tag headers are used for blocks with 62 bytes of data or less and large tag headers can be used for any size block. A short tag header has 9 bits used for the block type and 6 bits used for a block length in number of bytes. If a block is 63 bytes or longer, it should be stored in a long tag, which consists of a short tag whose length is 0x3f and is followed by a 32-bit length. The following tables show the structure of both short and long tag record headers. A header is followed by its associated data.

SHORT HEADER – for records less than 63 bytes in length

FIELD	TYPE	COMMENT
Tag	UB[10]	Tag id
Length	UB[6]	Length of tag

Note that the short header is not a 10 bit field with the tag followed by a 6 bit length. It is actually a 16 bit word with the high order 10 bits having the tag and the low order 6 bits having the length.

LONG HEADER– for records 63 bytes in length or greater

FIELD	TYPE	COMMENT
Tag	UB[10]	Tag id
Long Header Flag	UB[6]	Always 0x3F
Length	UI32	Length of tag

Definition tags are used to add character information to the dictionary. Definition tags define data that can be reused in a file like shapes, bitmaps, buttons, text, fonts and sounds. Control tags manipulate characters and provide administrative functions. For example, control tags are used to move objects on the screen, show frames, and start sounds.

Tags can occur in the file in any order. Definition tags must occur in the file before any control tags that refer to the character that is defined. The order of control tags defines the order of the animation. This ordering of tags is how the SWF file

format supports streaming. The rule is that a given tag should only depend on tags that come before it. A tag should never depend on a tag that comes later in the file. The End tag should be used at the end of the file to indicate the file's end.

5 In order to support streaming and references between tags, SWF uses the concept of characters and a dictionary of characters that have been encountered so far in a file. A character has a 16-bit numeric name. Every character in a file must have a unique name. Typically, the first character is #1, the second character is #2 and so on. Character #0 is special and considered a null character. Characters are entered into the dictionary when a define tag is encountered in the .swf file. Characters are
10 looked up in the dictionary based on the character identification number.

SWF uses integers and fixed point numbers for all of its coordinates and transformation matrices. The coordinate space is defined by integer coordinates. This coordinate space can be mapped to the display in any way that a player might want. By convention, coordinates are specified where one unit equals 1/1440 inches. On a
15 typical 72 dpi display, this is equivalent to saying that a single pixel is divided into 20 units. This choice of units provides a good compromise between giving a high resolution for zooming in and precise placement of objects. A higher resolution would increase the file size because of the extra bits needed for precision.

The model for processing a stream is that all tags are processed in a stream
20 until a ShowFrame tag is encountered. At that point, the display list is copied to the screen and the player is idle until it is time to process the next frame. A SWF file is divided into numbered frames by ShowFrame tags. Frame 1 is defined by performing all the control operations before the first ShowFrame tag. Frame 2 is defined by performing all the control operations before the second ShowFrame tag and so on.

25 The discussion above describes the input to the Presentation Server and the output from the Presentation Server. More details of the operation of the Presentation Server are now provided below.

Figure 4 is a block diagram of one exemplar architecture for the Presentation
30 Server. The main entry point to the Presentation Server is Network Adaptor 160, which is used to connect to the host. In one implementation, Network Adaptor 160

can include an implementation of the standard JAVA HttpServlet interface. At initialization time, this servlet receives configuration parameters from Configuration Manager 162. Configuration Manager 162 reads the configuration information from a local disk or other storage device. This information can include the desired logging (file names and level of status/error/debug logging), URL matching rules (patterns that describe how URLs map to local file system names), and authentication rules (patterns that describe which request require authentication). Configuration Manager 162 also keeps track of the plug-ins that are available for each of the plug interfaces in the presentation server. For example, Authenticator 168, discussed below, can include plug-ins for JDBC, LDAP, or other. Data Connection Manager 174 can include plug-ins for JDBC, SOAP, XML RPC, or other. Media Transcoder 176 can include interfaces for images (JPEG, GIF, BMP, PNG, other), audio (MP3, WAV, other), graphics (SVG, HTML, PDF, SWF, other), and video (MPEG, MPEG2, SORENSON, REAL, Animated GIF, other).

The Presentation Server listens for requests coming into the application server and/or web server via HTTP. As discussed, the Presentation Server can be within a web server, application server, or other system. The URLs for these requests are partially configurable at deployment time (some configuration via the HTTP Daemon and Servlet container, with additional configuration via Configuration Manager 162). Requests are handled by Request Handler 164. In one embodiment, a request can be for an application, data, service or other type of content. Each of these types of requests will be handled by Request Handler 164. Each request is associated with a session object, which is managed by Session Manager 166. Session Manager 166 can also access any one of a set of one or more Authenticators 168. In one embodiment, there can be one authenticator for each different type of authentication, for each type of content, or for each item of content. Each authenticator can be a plug-in, as described above. Request Handler 164 then communicates with Compiler 170 in order to compile the mark-up language descriptions of the requested content to object code. Compiler 170 communicates with Object Code Generator 172 to generate the object code. Request Handler 164 also communicates with Data Connection Manager

174 in order to receive data from or write data to external data sources. Data Connection Manager communicates with Object Code Generator 172 to generate object, which includes data received from these external sources.

Figure 4 also shows Media Transcoder 176 in communication with Compiler 170, Object Code Generator 172, and Data Connection Manager 174. Media Transcoder 176 is used to transform media from unsupported formats to supported formats. For example, the Presentation Renderer may only be able to utilize media in certain preset formats, for example, perhaps the Presentation Render only accepts images that are JPEG. Thus Media Transcoder 176 will transform all images from whatever format they are in to JPEG. The system will know the accepted formats for any particular renderer and will use Media Transcoder 176 to transform media to those accepted formats.

It is contemplated that the presentation server can be used with various presentation renderers, separately or concurrently. The example of the Flash players is just one renderer that can be used with the present invention. In one embodiment, there can be separate object code generators and compilers for each Presentation Renderer. In other embodiments, Compiler 170 and Object Code Generator 172 can provide object coding in various different formats. It is contemplated that when there are multiple Presentation Renderers, the requests received by the presentation server will include an indication of the Presentation Renderer so that the system will generate object code specific to that Presentation Renderer. In other embodiments, there will be a separate presentation server for each type of presentation renderer. Other configurations can also be used. In another embodiment, the compiler can be set up as two sets of components: a front end component and one or more back end components. The front end component is shared across all presentation renderers and will communicate to each back end component. There will be one back end component for each presentation renderer.

Figure 5 is a flow chart describing one embodiment of the operation of the presentation server of Figure 4. In step 200, an HTTP request is received at Network Adaptor 160. Protocols other than HTTP can also be used. In step 202, it is

determined whether this is an initial request from the particular client. If it is an initial request, then Request Handler 164 associates a session object with the client. If this is not an initial request, then the already existing session object for that client is accessed and associated with the request in step 206. The session object stores session specific state information, for example, the state of authentication. In step 208, it is determined whether authentication is required. If this request must be authenticated, then Session Manager 166 calls the Authenticator 168 that is currently registered with the session in step 210. Authenticator 168 attempts to authenticate in step 212. Authenticator 168 marks the status of the request as authenticated or not. In step 214, it is determined whether authentication is successful. If authentication is not successful, then an error message is returned to step 216. In other embodiments, the system may determine not to send an error message or may perform another act in response to a failed authentication. If authentication is successful, the process continues at step 220.

In one embodiment, the Presentation Server can respond to at least two types of requests: those that refer directly to mark-up language descriptions and those that refer to taking data from, sending data to, or managing data sources or web services. Based on whether the request is a data request or a request that refers to a mark-up language description, the method of Figure 5 performs different sets of steps (see step 220). In one implementation, mark-up language descriptions are used to provide an application. In some embodiments, a user would initiate a request for an application and that request would be used to access a mark-up language description of the application. When that application is running on the user's machine, the application may send a request for data. In one embodiment, an application can also send a request for another application. That is, a first application running on a client machine may send a request for a mark-up language description of a second application to be compiled and sent to the client.

If the request is for mark-up language description content, then (in step 226), Compiler 170, using Object Code Generator 172, accesses the mark-up language description and compiles it. In step 228, Request Handler 164 packages the compiled

code to create an HTTP response. In step 230, the HTTP response is sent to the client Presentation Renderer.

If the request was for data, then in step 234, Data Connection Manager 174 is accessed and provided with the request. In step 236, Data Connection Manager 174 connects to the appropriate external data source to have the data request performed by the external data source. The data request could include writing data to the data source, reading data from the data source, or managing the data (or data source). Step 236, therefore, can include receiving data back from the external data source. Step 236 can also include receiving a response to a data source operation. The data/information from the response is compiled into executable code in step 238, and the method continues as step 228. Note that the data can include media and step 238 could include accessing Media Transcoder 176 to transform the media to an acceptable format prior to insertion into executable code.

Figure 6 is a block diagram, which illustrates one exemplar architecture for implementing Compiler 170. Figure 6 show Compilation Manager 300 in communication with Pre-processor 302, Script Module 310, and data/view class/view instance Module 314. Modules 310 and 314 are communication with Scripts Compiler 312. Pre-processor 302 is also in communication with Cache 304. Compilation Manager 300 manages the compilation process. Pre-processor 302 is used to pre-process the mark-up language descriptions. Modules 310 and 314 are used with Scripts Compiler 312 to compile markup language descriptions. In one embodiment, Compilation Manager 300, Pre-processor 302, Script Module 310 and data/view class/view instance Module 314 are implemented in one object oriented class.

Figure 7 is a flowchart describing one embodiment of the process for compiling using the components of Figure 6 (see step 226 of Figure 5). In step 360, Compilation Manager 300 receives the request from the Request Handler 164. The content is accessed in step 362. That is, the compilation manager accesses the mark-up language description, which may be in a file, set of files or other type of container. In step 364, it is determined whether the mark-up language description is cached.

That is, when mark-up language descriptions are compiled, the compiled versions are kept in Cache 304. Pre-processor 302 will determine whether a particular mark-up language description is found in the cache (step 364). If the mark-up language description is cached, then the cache copy is returned to Request Handler 164 in step 366. Note that a mark-up language description can include other files by reference. Thus, Pre-processor 302 only returns a cached copy if all of the included files have not been changed since the cached copy was created. If the mark-up language description has not been found in the cache, then in step 368 Pre-processor 302 processes the included files. That is, the mark-up language description may include include-style statements, such as "#include file" or href-style includes. These statements imply that the text from the included file should be "in-lined" into the description/document before compilation. Files may be recursively included. In step 370, Compilation Manager 300 parses the mark-up language description in order to identify and distribute each of the elements to the appropriate module (e.g., modules 310 or 314). The elements include view instantiations, view class definitions, data connections and scripts. Module 310 is used for scripts. Module 314 is used for view instantiations, view class definitions, and data connections. Each module knows how to turn its element into object code by making use of scripts compiler 312 and object code generator 172. In many cases, the module will first translate its elements into script as an intermediate step for producing object code. The script is then compiled into object code. In step 374, the top level elements are distributed to the appropriate module, as described above. In step 376, these modules process the elements into object code. In step 378, Compilation Manager 300 assembles object code for each of the elements into a single executable. In step 380, that executable is added to Cache 304. In step 382, that executable is returned to Request Handler 164.

Figure 8 provides a flowchart describing one embodiment for compiling media into byte code. This process can be performed as part of step 376 or step 238. In step 452 a result set is received at the data connection manager from the data source. The result set could be a file or other type of container. The result set includes the media. In step 454, the result set is examined in order to find the media. Any media found in

the result set is added to a separate object for each media asset in step 456. The object would be similar to the objects described below with respect to other data. The object would include fields storing attributes such as the name of the media, the format, etc, as well as the actual media itself. In step 458, the actual media is removed from the object and replaced by a reference to the media. In step 462, the object (including the reference to the media) is compiled to SWF byte code in a similar fashion to other data described below. In step 464, it is determined whether the media data is in an acceptable format. For example, the Flash player described above accepts images in JPEG format. If the media is in a format that is not accepted by the player, then it is transformed using media transformer in step 466 to a format that is accepted by the presentation player. In step 468, the system creates a tag header for the media data. The tag header is in a format suitable for a SWF file, as described above. In step 470, the media data (not including the compiled object) are added to the tag, again in a format suitable for a SWF file. In one embodiment, it is contemplated that one tag will be used to store all of the media assets. Other embodiments use formats different than SWF and, therefore, would use formats other than SWF tags.

Figure 9 is a flowchart describing a process for compiling view instances. The process of Figure 9 will be performed during step 376. In step 520, the source code for the view instance including the tag name and attributes is accessed. In step 522, a script instruction is created that calls an instantiation view function. This instantiation view function is a function that examines the properties of its parameters and determines what kind of objects to create at run time. The instantiation view function also attaches children to the created objects. In step 524, the tag name and attributes are added to the source description of the object that is passed to the function. In step 526, it is determined whether the view instance includes any children that haven't been processed. If it does, then in step 528, a child including a tag name and attributes are added to the call to the instantiate view function. And the process continues at step 526. If there are no more children to process, then the instruction calling the instantiate view function, which is written in Action Script, is compiled to Action Script byte code. Compiling Action script to Action Script Byte code is

known in the art. In step 542, the byte code is added to an ACTION block. As an example, consider the following code for a view instance:

```
5 <window name="Contacts">
    <view x="5" y="10"/>
</window>
```

This code should be converted to the following action script source code:

```
10 LzInstantiateView(
    {tagName: "window", attrs: {name: "Contacts"},
    children:
    [{tagName: "view", attrs: {x: 5, y: 10}}]}
)
```

15 The function LzInstantiateView is the instantiate view function that is discussed above. The view instance being transformed has one child.

Figure 10 is a flowchart describing a process performed by the instantiate view function called by the instruction created in step 522. This function is performed when the byte code is executed by the Presentation Renderer on the client. In step 600, the instantiate view function receives an object based on the parameters of the function call. The example code above would cause the following object o (this is a metalanguage variable; there may not be a run time variable bound to the instance) to be created and passed to the instantiation view function:

```
25 o.TagName = "Window"
    o.attrs.name = "Contacts"
    o.children[0] = o1
    o1.TagName = "view"
30 o1.attrs.x = 5
```

```
o1.attrs.y = 10
o1.children = []
```

5 In step 602, the tag name for the object is accessed (e.g., o.TagName). In step 604, it is determined that there is a class identification (e.g. o.ClassName = "<user defined class name>"). If not, then a predefined instantiation function associated with the TagName is called, and that function is passed the attributes of the object in step 606. If in step 604 it is determined if there is a class identification, then it is assumed that the view is based on a user defined class and a function associated with the tag name is called, with the attributes of the object being passed as parameters in step 608. In step 612, it is determined whether the attributes of the object include the id attribute. If so, then in step 614, the global variable whose name is the value of the id attribute is set to the parent. Note that the id attribute identifies a unique name for the view instance. In step 620, it is determined whether there is another child to process. If there are no more children to process, then the method of Figure 10 is done. If there is another child to process, then in step 622, the instantiation view function is recursively called for that child. In step 624, the result of the instantiation view function the child is attached to the parent and the method continues at step 620.

10 The instantiation function called in steps 606 and 608 creates objects to be displayed in the user interface. In one embodiment, there is an instantiation function for each type of object being displayed. These instantiation functions call a constructor function to create the appropriate object. For example, there will be an instantiation function for windows, an instantiation function for buttons, etc. In one embodiment, there will be a table of instantiation functions for the predefined classes. 15 These functions are indexed by the TagName of the element. There can also be a separate table of instantiation functions for user defined classes (called in step 608) that are indexed by the name attribute. Note that the TagName for an element based on a user defined class is "instance." When a user creates a view class, this view class is compiled into a user-defined instantiation function, which creates an object at run time to be displayed in the user interface according to the user definition. This new 20 25 30

user-defined instantiation function is added to the above described table.

Figure 11 is a flow chart that describes a process for performing data connection requests. The process of Figure 11 is performed as part of step 236 of Figure 5. In step 700, the data connection request is received by the Data Connection Manager. Data queries can request to read data, write data or manage a database. When the data connection manager receives a request, it first determines the data connector object associated with the request (step 702) and the type of request—read, write or manage (step 704). In one embodiment, the Presentation Server provides a plug-in interface for adding new data connector objects. Each data connector object is able to communicate with a specific data source. The data connection request contains syntax that includes the name of the data connector object, and any data connector-specific arguments that further specify the request. The request may also include unambiguous syntax that identifies the data connection and query that has been previously defined in the current session. Below is an example URL for a JDBC connection request:

`http://host:port/LZ/myapp/?connector=id@source=id&...`

In step 706, the request information is provided to the appropriate data connector. In step 708, that data connector performs the request. Step 708 includes the data connector making a connection to the external data source and requesting that the external data source perform the operation desired. In step 710, the result provided from the external data source to the data connector is forwarded to the Data Connection Manager.

Figure 12 is a flow chart describing a process for compiling data to object code. The process of Fig 12 is performed as part of step 238 of Fig. 5. In step 818 of Fig. 12, the data connection manager accesses the data received from the data source. In step 820, the received data is placed in an XML document. For example, the following data may be received in response to a SQL SELECT statement:

id	name	phone
1	Joe Smith	4155551212

One embodiment of step 820 converts the above data to the following:

```
<resultset query="select id, name, phone from contacts where id=1">
<result id="1" name="Joe Smith" phone="4155551212">
</resultset>
```

In step 822, the XML is converted into ActionScript. For example, the above XML would be converted into the following:

```
resultSet0 = new Object();
resultSet0.name = 'resultSet';
resultSet0.attrs = new Object();
resultSet0.attrs.query = 'select id, name, phone from contacts where id=1';
resultSet0.children = new Object();
resultSet0.children.result0 = new Object();
resultSet0.children.result0.name = 'result';
resultSet0.children.result0.attrs = new Object();
resultSet0.children.result0.attrs.id = 1;
resultSet0.children.result0.attrs.name = 'Joe Smith';
resultSet0.children.result0.attrs.phone = '4155551212';
```

In step 822 of Figure 12, the ActionScript is compiled into ActionScript byte code. As described above, compiling action script into action script byte code is known in the art. In one embodiment, step 822 is skipped and the XML code is converted directly into byte code.

The foregoing detailed description of the invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit

5

the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. The described embodiments were chosen in order to best explain the principles of the invention and its practical application to thereby enable others skilled in the art to best utilize the invention in various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto.

101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278